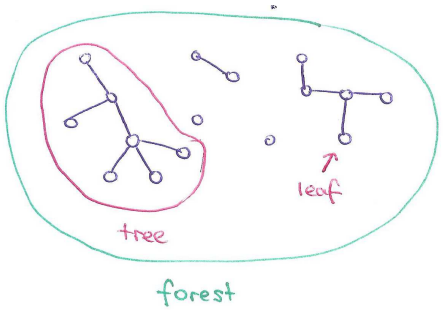
TREES

Reading: 4.1-2

- *acyclic* graph or *forest*: no cycles;
- *tree*: (nonnull) connected forest;
- *leaf*: degree 1 vertex.
- *trivial graph*: K_1.



(T1) Forests and trees are simple (loops, parallel edges give cycles) and bipartite (no odd cycles).

Lemma T2: A nontrivial tree has at least two leaves.

Proof: Let $P = v_0e_1v_1 \dots v_\ell$ be a maximal path (cannot be extended at either end) (A longest path would do.) Assume v_0 is not a leaf. Then it has another incident edge besides e_1, say e'. Since P is maximal, e' must join v_0 to some v_i, $i \geq 1$, and $v_0e_1v_1 \dots v_ie'v_0$ is a cycle, a contradiction. Hence v_0 is a leaf, and similarly v_ℓ is a leaf. ∎

Consequences: (T3) An acyclic graph has $\delta \leq 1$.

(T4) A graph with $\delta \geq 2$ has a cycle. (Contrapositive of (T3).)

(T5) Deleting a leaf from a connected graph G leaves a connected graph.

Proof: If G has a leaf then G is nontrivial. Deleting a leaf w does not affect the existence of a uv-path for $u, v \in V(G) - \{w\}$, so $G - w$ is still connected. ∎

Lemma T6: A tree T has $m = n - 1$.

Proof: By induction. If $n = 1$ then $T \cong K_1$ and the result holds. So suppose $n \geq 2$ and the result holds for trees with fewer vertices than T. By Lemma T2, T has a leaf w, and by (T5), $T' = T - v$ is still connected and hence still a tree. So, by the induction hypothesis $m' = n' - 1$. But $m' = m - 1$ and $n' = n - 1$, so $m = m' + 1 = n' = n - 1$, as required. ∎

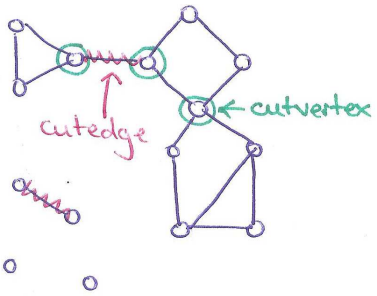
Consequences: (T7) If G is acyclic with c components, then $m = n - c$.

Proof: Add up both sides of $m_i = n_i - 1$ for all components $T_1, T_2, \dots, T_c$ of G. ∎

(T8) If G is acyclic and $m = n - 1$ then G is a tree.

Proof: By (T7), $c = 1$. ∎

- *cutedge* e: $G - e$ has more components than G;
- *cutvertex* v: $G - v$ has more components than G.



(T9) If we add edge e joining u, v to a graph, then the number of components goes down by 1 if u, v are in different components, and stays the same otherwise.

Reversing this, e is a cutedge if and only if u and v are in different components of $G - e$, and in that case $G - e$ has one more component than G,

Lemma T10: e is a cutedge $\Leftrightarrow$ e is not in a cycle.

Proof: A loop is never a cutedge, so we may assume e has two distinct ends u, v.

($\Rightarrow$) Suppose e is a cutedge. If e is in a cycle C then $C - e$ is a path joining u and v in $G - e$, contradicting (T9).

($\Leftarrow$) Suppose e is not in a cycle. If e is not a cutedge then there is a path P joining u and v in $G - e$. But then $P \cup e$ is a cycle containing e, a contradiction. ∎

Consequence: (T11) Every edge is a cutedge $\Leftrightarrow$ the graph is acyclic.

(T12) Every connected graph has a spanning tree.

Proof: Repeatedly remove an edge that belongs to a cycle. At each stage we are not removing a cutedge, by Lemma T10, so the graph remains connected. Continue until there are no cycles left. The result, T, is acyclic and connected and a spanning subgraph of G (we did not remove any vertices), i.e., a spanning tree of G. ∎

(T13) A connected graph with $m = n - 1$ is a tree.

Proof: By (T12), there is a spanning tree T'. Then by Lemma T6, $m' = n' - 1 = n - 1 = m$ so T' is the whole graph.

Theorem T14: For a nonnull graph T the following are equivalent.
(i) T is a tree, i.e., acyclic and connected.
(ii) T is connected and $m = n - 1$.
(iii) T is acyclic and $m = n - 1$.

Proof: (i) $\Rightarrow$ (ii) by Lemma T6. (ii) $\Rightarrow$ (iii) by (T13). (iii) $\Rightarrow$ (i) by (T8). ∎

From (i), (ii), (iii) any two of acyclic, connected, $m = n - 1$ imply the third.

(T15) Every connected graph on at least 2 vertices has at least two vertices that are not cutvertices.

Proof: Take leaves of a spanning tree.

Notation: If P is a path, uPv denotes the subpath from u to v.

Lemma T16: A graph G is a tree $\Leftrightarrow$ G is nonnull and loopless and $\forall\ u, v \in V(G)$ there exists exactly one uv-path (which we denote uGv).

Proof: ($\Rightarrow$) Suppose G is a tree, then G is loopless. Let $u, v \in V(G)$. Since G is connected there is at least one uv-path. Suppose that there are distinct uv-paths P_1, P_2. Let s be the vertex after which P_1 and P_2 differ, and let t be the next vertex of P_2 after s that also belongs to P_1. Then t cannot be before s on P_1 or P_2. So $sP_2tP_1^{-1}s$ is a cycle, a contradiction. Thus, there is a unique uv-path.

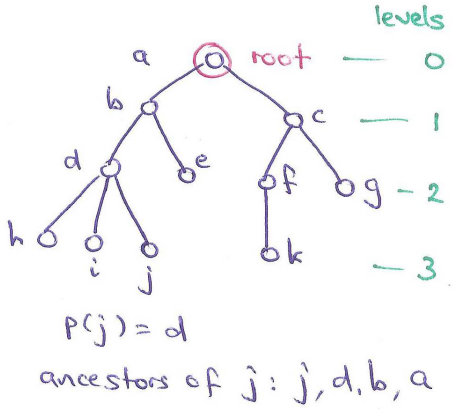
($\Leftarrow$) Suppose G is nonnull and loopless, and $\forall\ u, v \in V(G)$ there is exactly one uv-path. Then G is connected; we must show G is acyclic. Since G is loopless there are no cycles of length 1. If there is a cycle of length 2 or more, say $v_0e_1v_1 \ldots (v_\ell = v_0)$ then we have two distinct v_0v_1-paths, a contradiction. Thus, G is acyclic, as required. ∎
Need loopless!

Can write down two more characterizations of trees.

Theorem T17: For a nonnull graph T the following are equivalent.
(i) T is a tree.
(iv) T is connected and every edge is a cutedge.
(v) T is loopless and $\forall\ u, v \in V(T)$ there is exactly one uv-path in T.

Proof: (i) $\Leftrightarrow$ (iv) by Lemma T10. (i) $\Leftrightarrow$ (v) by Lemma T16. ∎

Rooted trees



Later will be convenient to pick particular vertex in tree and think of tree in relation to this vertex.

- r-*tree* or *tree rooted at* r is tree with special designated vertex r, the *root*.

In an r-tree there is a unique rv-path rTv.

- *ancestor of* v: any vertex of rTv (inc. v)
- *parent* $p(v)$: immediate predecessor on rTv (root has no parent)
- *proper* ancestor (not v itself), *descendant*, *related*
- *level* of v: $\ell(v) = d_T(r, v)$

Counting spanning trees

Won't spend much time on this. Just want to make the point that spanning trees can be counted fairly easily.

Cayley's Formula: There are n^{n-2} labelled n-vertex trees. Equivalently, K_n has n^{n-2} spanning trees.

For proof see B&M section 4.2.

Deletion-contraction formula: If $t(G)$ is the number of spanning trees of G and e is a link of G, then $t(G) = t(G - e) + t(G/e)$. [If e is a loop then $t(G) = t(G - e)$.]

Matrix Tree Theorem: Expresses number of spanning trees in any graph G as the determinant of a matrix obtained by modifying the adjacency matrix.

Next will look at some tree-based algorithms. First want to briefly discuss ideas from computational complexity.

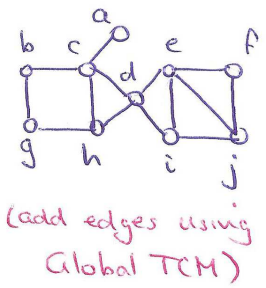
Tree construction methods

Want to construct spanning trees, maybe with given properties, reasonably efficiently. Can use to tell if graph connected, find components if not.

Read: 6.1.

Assume for now: G is connected.

Global Tree Construction Method (Global TCM): Start with edgeless spanning subgraph F. At each step choose an edge not forming a cycle (equivalently, joining two distinct components) with F and add it to F. When we cannot continue F is a spanning tree.



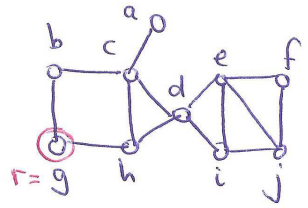
Could also call this 'Generic' TCM.

Know it will grow spanning tree: never creates cycles, decreases components by one each step so takes $n - 1$ steps, get acyclic graph with $n - 1$ edges: tree.

Notes: (1) At each step every component of F has at least one edge leaving it. So can choose a specific component for one end of edge.

(2) If apply to disconnected graph will build spanning forest consisting of spanning tree for each component. So can identify components. Can use to test if graph connected.

Local Tree Construction Method (Local TCM): Choose particular vertex r. Apply Global TCM, at each step adding an edge leaving the component containing r.



(add edges using Local TCM, r=g)

Notes: (1) At each step F has one component T containing r; all other components are isolated vertices. When we finish, $F = T$. Think of as method to grow T outward from root r.

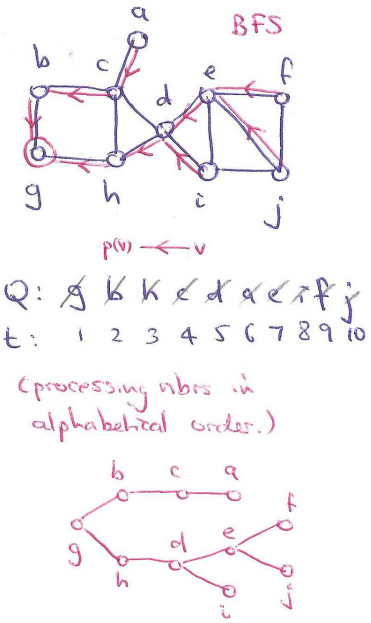
(2) At each step add uv with $u \in V(T)$, $v \notin V(T)$: then $u = p(v)$. So can build up parent function as build up tree.

(3) If we apply Local TCM to a disconnected graph, it finds all vertices reachable from r, i.e. T ends up a spanning tree of r's component.

Now look at two common implementations of Local TCM with useful properties.

Breadth-first search (BFS): implements Local TCM using a queue Q; tree stored using parent function p.

$Q = (r)$; mark r seen; now $= 1$; $t(r) =$ now;
for all vertices v, $p(v) = \emptyset$;
while Q nonempty {
  take x from front of Q;
  for each unseen neighbour y of x {
    $p(y) = x$; mark y seen; add y to back of Q;
    now $=$ now $+ 1$; $t(y) =$ now;
  }
}



Properties: (1) $\ell(v) = d_T(r, v) = d_G(r, v)$. So BFS computes distance from a given vertex for us.

(2) $|\ell(u) - \ell(v)| \leq 1$ for all adjacent u, v.

(3) Implementation requires $O(m + n)$ integer operations.

(1) and (2) proved in book; I won't do that.
So have poly. time algorithm to check if connected, find components, find distances.

Depth-first search (DFS): implements Local TCM recursively; tree stored using parent function $p(v)$. Can also implement non-recursively using a stack (see book although does not fully specify efficient implementation). Stores time vertex seen initially and finally in $t_I(v)$, $t_F(v)$.

now = 0;
for all vertices v, $p(v) = \emptyset$;
dfs-visit(r);

dfs-visit(v) {
    now = now + 1; $t_I(v)$ = now; mark v seen;
    for each neighbour u of v
        if u is unseen {
            $p(u) = v$; dfs-visit(u);
        }
    now = now + 1; $t_F(v)$ = now;
}

Important: Cannot write code as 'for each unseen neighbour u of v' because whether u is seen or not may change when other neighbours of v are visited.

Properties: (1) If v is ancestor of u in T, $[t_I(u), t_F(u)] \subseteq [t_I(v), t_F(v)]$. If u and v are unrelated, $[t_I(u), t_F(u)] \cap [t_I(v), t_F(v)] = \emptyset$. Gives efficient way to check if ancestor.

(2) If u, v adjacent in G then related in T.

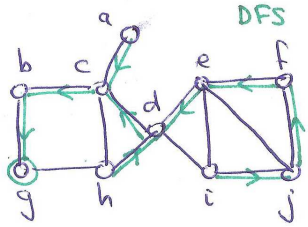
(3) Implementation requires $O(m + n)$ integer operations.

Again, I won't prove these.
Useful later for finding cutvertices in graph.

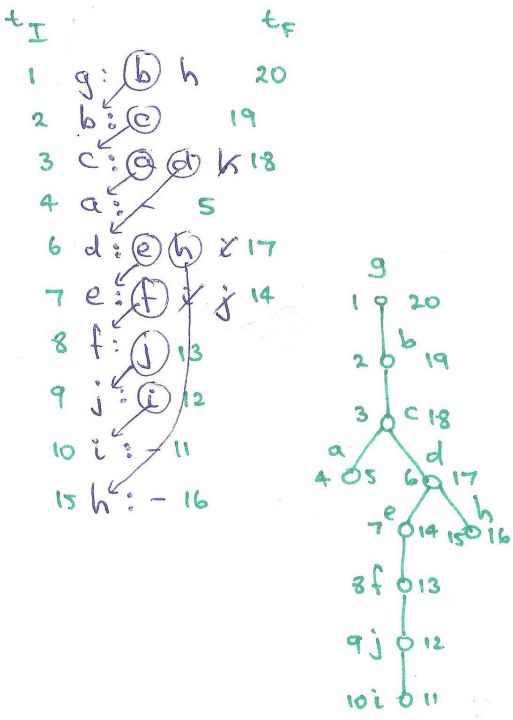
Alternative descriptions of BFS, DFS: Both use Local TCM, adding uv with $u \in V(T)$, $v \notin V(T)$.

BFS: choose u added to T as early as possible.

DFS: choose u added to T as late as possible.

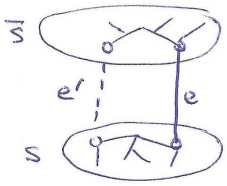


- process nbrs in alphabetical order
- put down currently unseen nbrs
- circle visited nbrs
- strike out nbrs that are seen when we look at them again



Want to look at min cost spanning trees, need theory first.

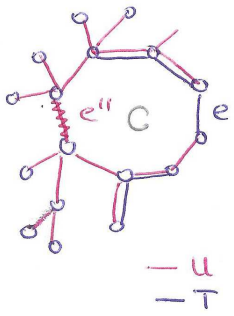
Read: 6.2, 8.5



Edge exchange properties: Let T, U be distinct spanning trees of a graph G, and $e \in E(T) - E(U)$.

(EE1) There is $e' \in E(U) - E(T)$ such that $T - e + e'$ is a spanning tree.

(EE2) There is $e'' \in E(U) - E(T)$ such that $U + e - e''$ is a spanning tree.



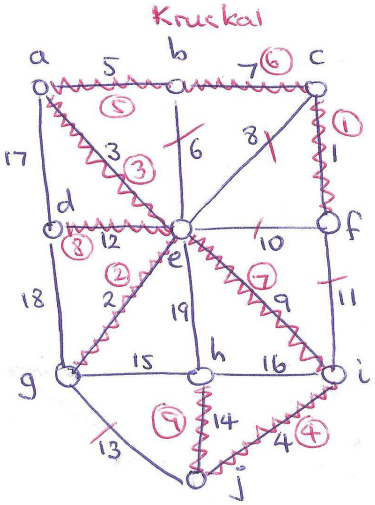
Proof: (EE1) e is a cutedge of T, so $T-e$ has two components with vertex sets S, $\overline{S} = V(G) - S$. Since U is connected it contains at least one edge e' joining S to $\overline{S}$ ($e' \neq e$). Since e is the unique edge of T joining S and $\overline{S}$, $e' \notin E(T)$. Then $T - e + e'$ is connected with $n-1$ edges, so it is a spanning tree.

(EE2) $U + e$ contains a unique cycle C formed by e and the unique path in U between the ends of e. Since T is acyclic, at least one edge e'' of C is not an edge of T ($e'' \neq e$). Now $U + e - e''$ is acyclic and has $n-1$ edges so it is a spanning tree. ∎

Minimum Weight Spanning Tree Problem: Given a connected graph G with nonnegative weights (or costs) $w(e)$ for each $e \in E(G)$, find a spanning tree T so that $w(T)$ (sum of weights of edges of T) is minimum.

Obvious applications to minimum cost networks. Min cost spanning tree is min cost connected network since minimal connected networks are spanning trees.

Kruskal's Algorithm: Apply Global TCM, being greedy, i.e., picking an available edge of minimum weight at each step.



Proof this works: Suppose Kruskal constructs T by choosing edges $e_1, e_2, \ldots, e_{n-1}$ in that order. Let U be an min. weight spanning tree that also maximizes the value of k such that U contains $e_1, e_2, \ldots, e_k$. We claim that $U = T$, i.e., $k = n-1$. Assume for a contradiction that $k < n-1$.

Then $e_{k+1} \in E(T) - E(U)$. By (EE2) there is $e'' \in E(U) - E(T)$ such that $U' = U + e_{k+1} - e''$ is also a spanning tree. Now $e_1, e_2, \ldots e_k, e'' \in E(U)$, so they do not form a cycle, so e'' was an available edge at the same time as e_{k+1}. Since Kruskal chose e_{k+1}, we must have $w(e_{k+1}) \leq w(e'')$. Then then $w(U') \leq w(U)$ so U' must also be a min. weight spanning tree. But U' contains $e_1, e_2, \ldots, e_k, e_{k+1}$, contradicting the choice of U.

Thus $k = n-1$, $U = T$, and so T is optimal. ∎

Easy to implement. First sort all edges by weight, go through in increasing order of weight adding any edge that does not create a cycle.

Important general idea: can apply greedy algorithm to get minimum weight 'cycle-free' object when exchange property applies. Leads to objects called matroids, generalize graphs in certain ways.

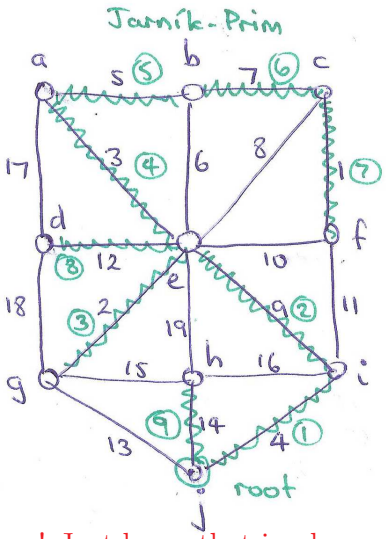
Notes: (1) If tie, choose any available min. wt. edge.
(2) Also works, modified suitably, for maximum wt. sp. tree.
(3) Don't actually need weights ≥ 0.

Jarník-Prim Algorithm: Apply Local TCM, being greedy, i.e., picking an available edge of minimum weight at each step.

Proof this works: Similar to proof of Kruskal; just choose e'' in slightly different way. Start off the same. Second paragraph:

Then $e_{k+1} \in E(T) - E(U)$. Let S_k be the set of vertices after adding $e_1, e_2, \ldots, e_k$; then e_{k+1} joins S_k to $\overline{S}_k = V(G) - S_k$. Now $U + e_{k+1}$ contains a unique cycle C, which must contain another edge e'' joining S_k to $\overline{S}_k$. We know that $e'' \notin \{e_1, e_2, \ldots, e_k\}$ (although possibly $e'' \in E(T)$). Since J-P chose e_{k+1} instead of e'', we must have $w(e_{k+1}) \leq w(e'')$. Now $U' = U + e_{k+1} - e''$ is acyclic with $n-1$ edges and hence a spanning tree.

Proof now concludes in the same way. ∎



Book gives complicated implementation with tentative weights, ignore! Just know that implementation can be done efficiently. Book also does alternative proof based on contracting edges.
Third method: Borůvka algorithm, *not* same as Kruskal as book claims.